Ghosts for Lists: from Axiomatic to Executable Specifications

Frédéric Loulergue[1], Allan Blanchard[2], and Nikolai Kosmatov[3]

[1] School of Informatics Computing and Cyber Systems
Northern Arizona University, Flagstaff, USA
frederic.loulergue@nau.edu
[2] Inria Lille — Nord Europe, Villeneuve d'Ascq, France
allan.blanchard@inria.fr
[3] CEA, List, Software Reliability and Security Lab, PC 174, Gif-sur-Yvette, France
nikolai.kosmatov@cea.fr

Abstract. Internet of Things (IoT) applications are becoming increasingly critical and require formal verification. Our recent work presented formal verification of the linked list module of Contiki, an OS for IoT. It relies on a parallel view of a linked list via a companion ghost array and uses an inductive predicate to link both views. In this work, a few interactively proved lemmas allow for the automatic verification of the list functions specifications, expressed in the ACSL specification language and proved with the Frama-C/Wp tool.
In a broader verification context, especially as long as the whole system is not yet formally verified, it would be very useful to use runtime verification, in particular, to test client modules that use the list module. It is not possible with the current specifications, which include an inductive predicate and axiomatically defined functions. In this early-idea paper we show how to define a provably equivalent non-inductive predicate and a provably equivalent non-axiomatic function that belong to the executable subset E-ACSL of ACSL and can be transformed into executable C code. Finally, we propose an extension of Frama-C to handle both axiomatic specifications for deductive verification and executable specifications for runtime verification.

Keywords: linked lists, executable specification, deductive verification, runtime verification, Frama-C, internet of things

1 Introduction

Among distributed systems, connected devices and services, also referred to as the Internet of Things (IoT), have proliferated very quickly in the past years. There are now billions of interconnected devices, and this number is growing. It is anticipated that by 2021, about 46 billion devices will be in use.

Some of these devices are in service in security-critical domains, but even in domains that are not necessarily critical, privacy issues may arise with devices collecting and transmitting a lot of personal information. Moreover, insufficiently secured devices can be used, for example, for massive distributed denial of service

attacks [8]. This raises important security challenges. Formal methods – that have been successfully used for years in highly critical domains – can help today to bring security into the IoT field.

While the correctness of an implementation with respect to a formal functional specification provides a very strong form of guarantee, it can be very costly to achieve, and is currently mostly reserved to domains where it is required by regulations or offers a competitive advantage. In practice, it is very useful to rely on a combination of formal methods to achieve an appropriate degree of guarantee: static analysis to ensure the absence of runtime errors, deductive verification to prove functional correctness, and runtime verification for parts of code that cannot be (or are not yet) proved using deductive verification.

This work uses Frama-C [7], a framework for source code analysis of industrial-size programs written in C. Frama-C offers combined formal methods approaches, by providing its users with a collection of plugins that perform static and dynamic analysis for safety and security critical software. Collaborative verification across plugins is enabled by their integration on top of a shared kernel, and their compliance to a common specification language: ACSL [1].

Recently, Frama-C has been applied in the context of the IoT for verification of several modules of Contiki [5], an open-source operating system for the IoT. In our previous work, we formally verified the linked list module of Contiki [2]. The verification technique relies on companion ghost arrays to provide an alternative view of the lists and a *linking predicate* relating a list and its companion array. A small set of lemmas (proved using Coq [12]) allow us to verify the specifications of the list module functions automatically using the Frama-C/Wp tool.

In a broader verification context, especially when some parts of the system are not yet proven, it would be desirable to benefit of the formal specification of the proven module while testing its (yet unproven) client modules, e.g. to check that the preconditions of proven functions are always satisfied along the tests of a client module. A Frama-C plugin, called e-acsl2c in this paper, can automatically transform specifications into executable C code, verifiable at runtime, but only if they belong to the executable subset of ACSL, named e-acsl [4, 11]. In our work [2], the formal specification of the list module relies on an inductive linking predicate and an axiomatically defined function, which are convenient for deductive verification but do not belong to this subset. Of course, we do not want to loose the effort put in conducting deductive verification.

This early-idea paper explores a solution inspired by verification by program transformation: instead of generating an executable definition from an axiomatic one, we propose to define an executable one and prove its equivalence with the axiomatic definition. To support this methodology, the already implemented e-acsl2c tool could be extended as follows: e-acsl2c would look for known equivalences when encountering a non-executable element in order to produce its executable counterpart. This extension is simple enough to be quickly and safely added to e-acsl2c, but of course requires the user to state the executable definitions and prove their equivalence with the axiomatic ones. We present a proof-of-concept application of this approach to the list case study [2].

The remaining part of the paper is organized as follows. In Section 2 we give an overview of Frama-C, and its Wp and e-acsl2c plugins. Section 3 briefly presents the verification of the Contiki list module, with a particular emphasis on two axiomatic definitions that cannot be handled by e-acsl2c. We then present equivalent executable specifications and discuss the proofs of equivalence with the axiomatic specifications (Section 4). In Section 5 we discuss an extension of e-acsl2c to support such an approach and related work.

2 Frama-C Platform and its WP and E-ACSL2C Plugins

Frama-C [7] offers various plugins built around a kernel providing basic services required for any analysis. It relies on the CIL frontend [9] extended to treat acsl annotations. acsl, for ANSI/ISO C Specification Language, is based on the notion of contract like in Eiffel or JML. It allows users to specify functional properties of programs through pre/post-condition, and provides different ways to define predicates and logic functions. Some useful built-in predicates and logic functions are provided, to handle for example pointer validity or separation.

Wp is a deductive verification plugin provided with Frama-C. It is based on weakest precondition calculus. Given a C program annotated in acsl, Wp generates the corresponding proof obligations that can be discharged by SMT solvers or with interactive proof. A combination of automatic and interactive proofs often offers a good trade-off for a complete proof. Indeed, some properties can only be defined recursively, and in this case, SMT solvers often become inefficient, trying to unroll them. By using inductive or axiomatically defined functions, we can prevent this behavior but reasoning about them still requires induction, a task that SMT solvers are not good at. Thus, the last step is generally to state lemmas that can be directly instantiated by SMT solvers. These lemmas can be easily used by SMT solvers to verify specifications, but their proofs require to reason by induction: they are proved interactively.

The e-acsl2c plugin transforms annotations that belong to the executable subset e-acsl of acsl into C code in order to verify them at runtime [4]. This subset [4, 11] restricts acsl to executable features: quantifications over finite intervals only, no axioms or lemmas, no inductive predicates or axiomatic definitions of logic functions. Mathematical (unbounded) integer arithmetic is supported via a translation to larger types or using a dedicated library (GMP). Pointer properties (such as validity) are handled thanks to a dedicated memory model [14].

3 The List Module of Contiki

The linked list module of Contiki is a critical module of the kernel intensively used in the core part of the OS. Its formal verification was thus necessary. The formal verification we have performed [2] relies on a view of each list via a ghost array that mirrors it, and the following *linking predicate* defining their relation.

```
1 inductive linked_n{L}(struct list *root, struct list **cArr,
2                       ℤ index, ℤ n, struct list *bound){
3 case linked_n_bound{L}:
4 ∀ struct list **cArr, *bound, ℤ index ;
5   0 ≤ index ≤ MAX_SIZE ⇒
6     linked_n(bound, cArr, index, 0, bound);
7 case linked_n_cons{L}:
8 ∀ struct list *root, **cArr, *bound, ℤ index, n ;
9   0 < n /\ 0 ≤ index /\ 0 ≤ index + n ≤ MAX_SIZE /\
10  \valid(root) /\ root == cArr[index] /\
11  linked_n(root->next, cArr, index + 1, n - 1, bound) ⇒
12    linked_n(root, cArr, index, n, bound);
13 }
```

This predicate inductively relates a list starting at `root` to a segment of companion array `cArr`, starting from an offset `index` and having `n` elements, that ends with the excluded cell address `bound` (either `NULL` or a pointer to the first non-represented list element if any). This relation is verified (cf. axiom `linked_n_cons`, lines 6–11) if `root` is a valid memory location, if we find this value at offset `index` of `cArr`, and if, recursively, the list that starts at `root->next` is linked to the segment starting from `index+1` with `n-1` elements. That is, for all i, the address of the i^{th} cell of the list can be found at `index+`i of `cArr`. The empty list (cf. axiom `linked_n_bound` lines 3–5), that starts and ends with `bound`, is related to a `cArr` segment from any `index` for a length of `0` elements. The linking relation between the list and its ghost array is maintained as an invariant by the functions of the list API. Thus, for verification we add some ghost code that updates the companion array when needed.

Some lemmas allow us to split (or merge) a list into sub-lists related to consecutive subranges of the companion array. It allows, for example, to prove properties about the removal of an element of the list, where we have to show that the beginning of the list did not change, and that all elements starting from the item to remove have been shifted, so the list does not contain the item anymore. Of course, that means that we need a way to specify the location of an element in the list. This is done using the `index_of` function presented below:

```
1 axiomatic Index_of_item {
2   logic ℤ index_of(struct list *item, struct list **cArr,
3                    ℤ down, ℤ up) reads cArr[ down .. up-1 ];
4   axiom no_more_elements:
5   ∀ struct list *item, **cArr, ℤ d, u ; 0 ≤ u ≤ d ⇒
6     index_of(item, cArr, d, u) == u;
7   axiom found_item:
8   ∀ struct list *item, **cArr, ℤ d, u ;
9     0 ≤ d < u /\ cArr[d]==item ⇒ index_of(item,cArr,d,u)==d;
10  axiom not_the_item:
11  ∀ struct list *item, **cArr, ℤ d, u ;
12    0 ≤ d < u /\ cArr[d]≠item ⇒
13    index_of(item,cArr,d,u) == index_of(item,cArr,d+1,u);
14 }
```

This function searches an `item` in the companion array `cArr` (therefore, in the list), between two indices `down` and (excluded) `up` and returns the corresponding offset. If the element is not in the list, the function returns `up`. This definition is also recursive. For an empty range, `down` equals `up`, the offset is `up` (cf. axiom `no_more_elements`, lines 5–6). For a non-empty range, if the element is the first one (cf. axiom `found_item`, lines 7–9), the offset is the index of this element `down`. Finally, for a non-empty range, if the first element, at offset `d`, is not the one we are searching (cf. axiom `not_the_item`, lines 10–12), the function has to search the item in the subrange that starts at `d+1`.

Finally, additional properties (cf. [2]) are required to specify memory separation between the different elements of the list and between the elements of the list and the ghost array. For lack of space, we do not present them here.

4 From Axiomatic to Executable Specifications

We design new specifications in the E-ACSL subset of ACSL, and prove their equivalence with the axiomatic specifications presented in the previous section.

An executable linking predicate `linked_exec` equivalent to `linked_n` follows:

```
1 logic boolean array_view(struct list *root,
2                          struct list **cArr,
3                          ℤ idx, ℤ size, struct list *bound) =
4   (size==0)? root==bound : (root==cArr[idx] ∧
5     array_view(root->next, cArr, idx+1, size-1, bound));
6
7 predicate linked_exec{L}(struct list *root,
8                          struct list **cArr, ℤ idx,
9                          ℤ size, struct list *bound) =
10  0 ≤ size ∧ 0 ≤ idx ∧ idx + size ≤ MAX_SIZE ∧
11  (∀ ℤ k; idx ≤ k < idx + size ⇒ \valid(cArr[k])) ∧
12  array_view(root, cArr, idx, size, bound) == \true;
```

The idea is to replace an inductive predicate, which is not supported by E-ACSL, by a non-inductive predicate and a recursive logical function. Informally, tthe validity stated in `linked_exec` as a bounded quantification (line 9) and the equality between `root` and `cArr[idx]` imply the validity of `root` (as stated line 9 of `linked_n`) and the equality in `linked_n_cons`. The other conditions, in `linked_exec` and `linked_n` respectively, are identical.

An executable function almost equivalent to the axiomatic `index_of` follows:

```
1 logic ℤ index_of_exec(struct list *item, struct list **cArr,
2                       ℤ down, ℤ up) =
3   (down < 0 ∨ up < 0) ? -1 : (0 ≤ up ∧ up ≤ down) ? up :
4     (0 ≤ down ∧ down < up ∧ cArr[down] == item) ? down :
5        index_of_exec(item, cArr, down+1, up);
```

This function is not fully equivalent to the inductive axiomatic function `index_of` previously presented because the axiomatic definition says nothing when one of the bound is negative. An executable version could lead to runtime errors in that

case, thus it includes an additional check to prevent them. Therefore we add a new case to the axiomatic version `index_of` to ensure the equivalence with the new logical function and its (stricter) bound checks:

```
axiom invalid_bounds:
  ∀ struct list *item, **cArr, ℤ down, up ;
  (down < 0 ∨ up < 0) ⇒ index_of(item, cArr, down, up) == -1;
```

The deductive verification of the list module is not impacted by this modification.

We have proved the following two lemmas that state respectively that the axioms defining the function `index_of` and the recursive function `index_of_exec` are equivalent, and that the inductive predicate `linked_n` and the non inductive predicate `linked_exec` are equivalent:

```
lemma equiv_index_of:
  ∀ struct list *item, struct list **cArr, ℤ down, ℤ up;
  index_of(item, cArr, down, up) ==
  index_of_exec(item, cArr, down, up);
lemma equiv_linked:
  ∀ struct list *root, struct list **a, struct list *b,
  ℤ index, ℤ size;
  linked_n(root, a, index, size, b)⟺
  linked_exec(root, a, index, size, b);
```

The first lemma is proved using Coq as follows: after case reasoning to assure that $0 \leq$ `down` $<$ `up`, the result is established by induction on a value *len* equal to `up` $-$ `down` and replacing `down` with `up` $-$ *len*.

The second lemma is also proved in Coq. The first implication is proved by induction on the inductive predicate `linked_n`. The second implication is proved by induction on `size` and using two lemmas on `array_view` themselves proved by induction on `size`. The proofs are not very simple but comparable to some of the lemmas for `linked_n` proved for the deductive verification of the list API.

5 Discussion

The E-ACSL2C plugin currently does not support the full E-ACSL subset, but it is evolving rapidly. The proposed approach assumes a better support of the E-ACSL language to make the specifications described in the previous section executable.

Since this support is still partial, in order to apply the proposed approach, we proceed by manual transformation: as predicate definitions (such as `linked_exec`) are not supported yet, we inline them, i.e. instead of applying `linked_exec` in a specification, we copy its body into it. As logical function definitions are not supported yet, we define corresponding C functions (and specify their equivalence with their logical counterparts, these specifications being automatically checked by Wp) and hard-coded calls to these functions in the body of the C list functions we specified, using C `assert`. Our study provides a proof-of-concept for the proposed approach of creating a (provably equivalent) executable specification.

Note that even if the `linked_exec` predicate is proved to be equivalent to the previous `linked_n` predicate, the inductive version is still needed for the formal proof to ensure that we get an induction principle on the Coq side, but also to prevent the SMT solvers or WP to unfold the predicates during automatic proof.

Assuming the E-ACSL specification language is fully supported, we envision an extension of E-ACSL2C to support this approach in a convenient and semi-automatic way. We propose a new annotation `equivalent` taking two names as arguments (either two predicates or two functions), that would:

- generate the corresponding equivalence lemma (to be proved),
- make E-ACSL2C replace the first argument with the second argument for the generation of C code.

Note that this annotation could be transitive: if the second argument is still not executable, the system could look for another equivalence relating the second name to a third one, and so on.

The proposed approach enables combined verification, very helpful in practice as many real-life systems are not fully proved. Its benefit is the possibility to use both axiomatic specification, more convenient for deductive verification, and equivalent executable specification, usable for dynamic verification, without the need for a large and complex extension of E-ACSL2C. For more automation – that would require much more implementation work – it could be possible to build on the work of Tollitte et al. [13].

For higher-level languages, such as Eiffel or Java, a related approach is to associate a model to a class [10]. The model plays the same role as the companion array in our approach. Such a model is also a class, but an immutable one used only for verification purposes. Model classes are valid classes of the considered language, and can therefore be used in dynamic verification tasks. For deductive verification purposes, the classes may be translated to elements of theories of the underlying theorem provers. In this case, the faithfulness of the mapping may be checked [3].

As future work we also plan to experiment with alternative specifications in the spirit of the work of Gladisch and Tyszberowicz [6]. In the case of JML, they used a pure observer method that takes a list object and an index, and returns the object at that index in the list, to specify Java methods on a linked list data structure. While the methods they consider are simpler than the list API of Contiki, our ghost arrays can essentially be seen as observations of the linked lists. We could consider such an observer directly written as a logical function in E-ACSL. C pointers are however not Java references and can lead to some complications.

Acknowledgment. This work was partially supported by a grant from CPER DATA and the project VESSEDIA, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. The authors thank the FRAMA-C team for providing the tools and support. Many thanks to the anonymous referees for their helpful comments.

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
2. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In: Proc. of the 10th NASA Formal Methods Symposium (NFM 2018). LNCS, vol. 10811, pp. 37–53. Springer (2018)
3. Darvas, Á., Müller, P.: Proving consistency and completeness of model classes using theory interpretation. In: Proc. of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010). LNCS, vol. 6013, pp. 218–232. Springer (2010)
4. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proc. of the ACM Symposium on Applied Computing (SAC 2013). pp. 1230–1235. ACM (2013)
5. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A lightweight and flexible operating system for tiny networked sensors. In: Proc. of the 29th Annual IEEE Conference on Local Computer Networks (LCN 2004). pp. 455–462. IEEE Computer Society (2004)
6. Gladisch, C., Tyszberowicz, S.S.: Specifying linked data structures in JML for combining formal verification and testing. Sci. Comput. Program. 107-108, 19–40 (2015)
7. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015)
8. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.M.: DDoS in the IoT: Mirai and other botnets. IEEE Computer 50(7), 80–84 (2017)
9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of c programs. In: Proc. of the 11th International Conference on Compiler Construction (CC 2002). pp. 213–228. Springer (2002)
10. Polikarpova, N., Furia, C.A., Meyer, B.: Specifying reusable components. In: Proc. of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010). LNCS, vol. 6217, pp. 127–141. Springer (2010)
11. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, http://frama-c.com/download/e-acsl/e-acsl.pdf
12. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr,
13. Tollitte, P., Delahaye, D., Dubois, C.: Producing certified functional code from inductive specifications. In: Proc. of the 2nd International Conference on Certified Programs and Proofs (CPP 2012). pp. 76–91 (2012)
14. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proc. of the International Symposium on Memory Management (ISMM 2017). pp. 47–58. ACM (2017)